Name Trees: Uniform and Extensible Interactions Between Languages and Language Extensions

Filip Strömbäck
Department of Computer and Information Science
Linköping University
Linköping, Sweden
filip.stromback@liu.se

ABSTRACT

Storm is a system for creating extensible languages. It allows creating both language extensions and entirely new languages conveniently. This is useful to create domain-specific languages (DSLs) that make certain specific tasks easier to achieve, and to add new functionality to existing languages. This extensibility is provided partially by the extensible syntax provided by Storm, and partially by the *name tree* that represents the global namespace in the system. This paper focuses on the name tree, which is shared between all languages in the system and allows them to exchange information between each other seamlessly. The name tree is extensible, which makes it possible for languages to extend the representation in order to accommodate new concepts while retaining compatibility with existing languages.

This paper describes the name tree used in Storm and illustrates how it can be used to incorporate new concepts in a type-safe way. We do this by showing extensions to the imperative language Basic Storm that add support for type-safe SQL queries and type-safe algebraic effects. We also show how the name tree allows Basic Storm to interact in a type-safe way with a separate language that is used to define grammars and semantic actions. The name tree is also useful for run-time reflection, and we will show how the information in the name tree can be used to implement a visual debugger and other tools that help developers understand the behavior of the system.

CCS CONCEPTS

• **Software and its engineering → Extensible languages**; *Runtime environments.*

KEYWORDS

extensible languages, domain-specific languages, lazy compilation, name trees, Storm

ACM Reference Format:
Filip Strömbäck. 2024. Name Trees: Uniform and Extensible Interactions Between Languages and Language Extensions. In *Proceedings of PX/24.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

PX/24, Mar 11, 2024, Lund, Sweden

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Domain-specific languages (DSLs) are useful in many situations [7]. As an example, many popular GUI frameworks provide a DSL for specifying the layout of components more conveniently compared to using a general purpose programming language. Two examples of frameworks that take this approach are GTK[1] and Android[2]. Both of them provide a DSL based on XML for the purposes of specifying layout. However, since the DSL is mostly separate from the host language, the developer needs to connect the two together manually. This makes it more cumbersome than necessary to use the DSL and it increases the risk for mistakes since the system is not able to type-check the interactions between the languages.

Another approach is used by frameworks like React.[3] They instead provide an *embedded* DSL as a language extension that makes it possible to specify layout alongside the rest of the application. This approach is more costly to implement, since the compiler for the DSL needs some level of awareness of the host language to be able to compile the DSL properly. It does, however, allow the DSL to integrate more seamlessly with the host language, which eliminates the need for the programmer to connect the two manually and allows type-checking.

A drawback of embedded DSLs is that they are often implemented as a preprocessor that transforms it into equivalent code in the host language. Since many preprocessors only accept a single DSL, it is often not possible to use multiple different DSLs in the same source file. There are, however, systems like SugarJ [3], ableJ [19], and ableC [10] that provide a preprocessor that can be extended to support multiple DSLs in a modular fashion. A problem that remains with the above-mentioned systems is that they are compiled to source code in the host language. This means that they are limited by the constraints imposed by the host language. As such, it is difficult for a DSL or language extension to extend the type system of the host language, especially if it is desirable to allow other extensions to use or further extend the new concepts in the future.

These issues are all addressed in the extensible system Storm [16] using a *name tree*. Storm uses its name tree to provide a global shared namespace where all languages and language extensions running in the system are able to store and look up *named entities* using an uniform interface. These named entities can represent standard concepts like types and functions, but it can also be extended with concepts that are unique to individual languages or DSLs, such as grammars or database schemas. In Storm, the shared

[1] https://docs.gtk.org/gtk4/class.Builder.html
[2] https://developer.android.com/develop/ui/views/layout/declaring-layout
[3] https://react.dev/learn/writing-markup-with-jsx

name tree is also available to programs running in the system. This makes it possible to use the name tree for reflection, and even to modify the system at run-time. This, in turn, allows languages and language extensions to be implemented in any language that is supported by Storm. It also allows building powerful, language agnostic development tools easily.

The remainder of this paper describes how Storm utilizes its name tree to make it easy to develop both embedded and standalone DSLs that interact with other languages in the system in a type-safe way (Sections 2 and 4). We will also show how DSLs and language extensions can extend the representation in the name tree with representations of new concepts that are usable by the host language and also accessible to future extensions (Section 5). Finally, we will show a few examples of tools that utilize the representation in the name tree to help developers debug issues and to explore libraries and languages in the system (Section 6).

2 STORM

Storm itself is not a programming language. Rather, it consists only of a runtime system for languages to use and a collection of tools to aid development of extensible and interactive programming languages [16]. Storm does, however, include two languages by default: *Basic Storm* and *the Syntax Language*. The former is a general purpose imperative programming language designed to closely reflect the internal representation used in Storm. The latter is a DSL for defining grammars used to parse extensible languages. Even though these languages are included by default, they are not treated specially by the system in any way. They are simply included as a convenient starting point for further development of programming languages and language extensions.

To aid the development of programming languages and language extensions Storm provides a *runtime system*, a library for in-process *code generation*, a *parser* for extensible languages, and a *name tree* that languages can use to resolve names in a uniform manner. These are shown in Fig. 1. Combined, these components allow a great level of flexibility. For example, the name tree is used to expose the functionality of all the components in Storm to all languages in the system. Furthermore, the in-process code generation alongside the uniform name resolution provided by the name tree makes it possible to implement languages or language extensions in any language supported by the system. In particular, this means that it is possible to develop DSLs specifically for making development of future DSLs or languages easier [16]. The in-process code generation also allows modifying the system while it is running, which makes it possible to implement techniques like *dynamic software updating* [9, 18] to allow live programming by making it possible to make changes to running programs.

The remainder of this section provides an overview of the aforementioned components, and a brief description about the interactivity of the system. The name tree, which is the focus of this paper, is covered in greater detail in Section 4. Storm is freely available at `https://storm-lang.org/` and works on both Windows and Linux.

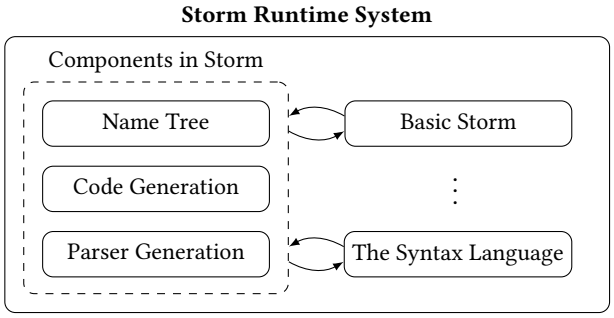

Figure 1: Overview of Storm. Adapted from [16].

2.1 Runtime System

The runtime system provides fundamental services to all code that is running in the system. Most importantly, the runtime system provides a garbage collector for memory management (currently, the MPS [1], but it is possible to use other collectors as well). It also manages all threads in the system and implements a user-mode scheduler to support green threads for languages that uses them. For example, Basic Storm uses green threads to implement message passing between threads. This is, however, outside the scope of this paper.

Apart from these low-level system facilities, the runtime system provides a number of primitive data types (e.g., integers and strings) and containers (e.g., arrays and hash tables) through the name tree. These types are intended to provide a set of types that languages can use to exchange information in a standardized format. Languages are, however, free to implement their own primitive types and containers if they desire. This would, however, make it more difficult for different languages to interact.

2.2 Code Generation

Storm provides a library that allows languages to generate executable code in a portable manner. The library is able to emit code for x86, x86-64, and ARM64 from the provided intermediate representation. The code is generated in-process, which immediately makes it usable by the program. This is important for the flexibility of the system as it removes the distinction between compile-time and run-time, which is one part of why it is possible to use any language in the system to implement new languages and extensions. For example, even though it is not currently possible to pre-compile Storm programs, it is possible to implement parts of Basic Storm in itself (e.g., array literals, string interpolation, and lambda functions).

The code generation library also allows dynamic re-linking of the running program. This makes it possible to replace the implementation of functions in the program at run-time. Storm currently uses this ability to implement *lazy compilation*, which means that most functions are not compiled until they are called for the first time. This ability is also used to allow updates to running programs, as described in Section 3.

2.3 Parser Generation

To aid languages in parsing their input, Storm also provides facilities for generating extensible parsers that support embedded DSLs.

As described in a previous paper [16], the parser generation facilities dynamically consume productions and non-terminals from the name tree and construct parse tables on the fly based on the grammar the language and/or the user wishes to use. Since the name tree is hierarchical, this makes it possible to structure the grammar as modules that can be imported as if they were libraries. This works similarly to the embedded DSLs implemented by SugarJ [3] and ableJ [19], but the underlying mechanisms differ. Each production is also associated with a semantic action (called *syntax transforms* in Storm) to allow extensions to specify the semantics for the new syntax without modifying the implementation of the host language.

In order to not restrict languages and language extensions, the facilities provided by Storm supports all context-free grammars, even ambiguous ones. As such, Storm currently generates parse-tables for a GLR [13] parser. Furthermore, the input is not tokenized in a separate step. Rather, the syntax representation (and the Syntax Language) allows specifying the tokenization in the grammar, and tokenization is then driven by the parser. This makes it possible for language extensions to customize the tokenization, for example by providing alternate syntax for string literals and/or comments in certain regions of the code.

As mentioned previously, the parser generation facilities operate on a syntax representation stored in the name tree. As with Basic Storm, the Syntax Language is simply a language that produces this representation that is included in the system by default. As such, it is possible to use the parser generation facilities with other languages that produce the same representation. It is also possible for other parts of the system to consume the generic representation for other purposes. For example, Storm contains a library for generating simpler but faster recursive descent parsers from the same representation. Finally, since the Syntax Language is implemented in itself, it is also possible to extend the syntax language using itself.

3 INTERACTIVITY

Storm aims to provide a convenient and interactive development experience. As we shall see (section 4.4), the compilation process is managed entirely by Storm, meaning that running a program is usually as easy as typing `storm <path>` where `<path>` is the location of the source code. It is also possible to run Storm interactively. Storm can either be launched stand-alone, in which case it provides an interactive top-loop. It is also possible to launch Storm as a language server, which allows an editor (currently only Emacs) to utilize Storm for syntax highlighting, accessing documentation, and interacting with the running program.

The ability to dynamically re-link programs (see Section 2.2) is also utilized to allow updating running programs. Presently, the possible updates are limited to adding new definitions and updating the body of existing functions. Preliminary work on extending these capabilities to also allow updating instantiated data structures exists as an experiment, but is not yet complete. This preliminary work does, however, indicate that a larger set of updates are possible.

As is the case with the other components in the system, these abilities are exposed to programs and languages in the system. It is thus possible to write programs that utilize the (currently limited) ability to update code to partially reload themselves dynamically. For example, the presentation viewer that is bundled with Storm utilizes this ability to allow reloading presentation developed in an extension to Basic Storm without having to restart the entire application. The openness of the system also makes it possible to develop alternative language servers or IDEs in Storm itself.

4 THE NAME TREE IN STORM

This section describes the implementation of the *name tree* in Storm. As mentioned in Section 2, the name tree is the data structure that represents the global namespace in Storm that is shared between all languages in the system. Storm also uses the name tree to expose the functionality of the code generation and parser libraries to languages in the system. As such, the name tree is a central part of Storm, and a core contributor to its flexibility.

Since Storm is designed to be an interactive system, programs are executed in the same process as the compiler. This allows interleaving program execution with compilation, which in turn makes it possible to use any supported language to extend the system itself. As a part of this, Storm gives programs unrestricted access to the name tree. Programs may thus use the name tree for facilities like reflection, loading new code, or modifying their own execution environment. Not all programs require this ability, however. For such programs, it would be possible to save the output from the compilation process as a stand-alone binary to allow executing the program without a name tree. This is, however, not yet supported by Storm.

The previously cited paper about Storm [16] mainly focuses on the Syntax Language and how it is used to create DSLs as syntax extensions to host languages in Storm. It does, however, not cover the name tree in great depth. As such, this paper focuses on how the name tree is used to enable the syntactical extensibility presented in [16], and how it allows introducing new concepts in the system.

4.1 Named Entities

The name tree stores a set of *named entities* in a hierarchical fashion. Each named entity is represented by an instance of the class `Named`. These entities typically represent familiar concepts such as types, variables, or functions, and are represented as the subclasses to `Named` as described in Section 4.3. It is, however, possible to store any type of data in the name tree by creating custom subclasses to `Named`. The same mechanism can also be used to extend existing representations as we will see in Section 5.

Each named entity has a name in the form of a string and a list of zero or more parameters that each refer to a type in the name tree. This name is typically written in the form `f(a, b, ...)` where `f` is a string, and `a` and `b` are parameters that refer to types in the system. If the list of parameters are empty the parentheses are omitted entirely. Two entities are considered to have the same name only if the string and the list of parameters are equal. This makes it possible to implement generic data types and function overloading. For example, a type that stores an array of integers can be named `Array(int)`, which is different from the name `Array(float)`, used for an array of floating point numbers. Similarly, a function that accepts an integer parameter is named `f(int)`, while a version of the function that accepts a floating point number is named `f(float)`.

As previously mentioned, the name tree is hierarchical, much like a file system. This is achieved by using instances of the `NameSet` class, which inherits from `Named` and stores a set of named entities as children to itself. This entity is thus used as an internal node in the tree structure of the name tree and as the root of the entire name tree. In Storm, the `NameSet` type is used as a starting point for anything that may contain other entities, such as packages and types.

Since the name tree is hierarchical, it may contain multiple nodes with the same name that are located inside different `NameSet` nodes (c.f., files with the same name in different directories). As such, it is necessary to use an *absolute name* to identify nodes uniquely. Such names are typically written as a sequence of parts separated by periods (`.`). For example, the name `core.f(int)` is the name of a function named `f` that accepts an integer parameter. The function is located in a `NameSet` named `core` that is itself located in the root of the name tree. Note that each part of the sequence may contain parameters. For example, a function that is a member of a parameterized type may have the absolute name `Array(int).push(int)`.

4.2 Name Lookup in the Name Tree

The name tree also provides a standardized mechanism to look up names in the name tree. Given an absolute name, it is straight forward to traverse the name tree from the root and resolve each part of the name in succession. This approach is, however, inconvenient as it would require languages to use absolute names.

To avoid this, the name tree provides a representation of a *scope* in which names can be resolved. A scope consists of two things, firstly a reference to a `NameLookup` object that corresponds to the "current scope" in the name tree. As illustrated in Fig. 1, a `NameLookup` object represents a possibly unnamed element in the system that supports resolving names. All `NameLookup` objects do, however, have a pointer to a parent object that allows traversing the name tree towards the root. In addition to the reference to a `NameLookup` object, a scope also contains a strategy that describes how the name tree should be traversed to resolve a name. Typically, the strategy first attempts to look up the name relative to the `NameLookup` stored in the scope (i.e., looking in the current scope). If it is not found there, the name tree is traversed towards the root until a match is found.

The representation of scopes above does not only have the benefit that it can be customized by languages. The standardized representation also makes it possible for languages to exchange name resolution strategies. This, in turn, makes it possible for one language to embed a part of another language while making the embedded language follow the name resolution rules of the host language.

4.3 Standard Entity Types

To make interaction between languages easier, Storm defines several entity types that represent concepts that are commonly found in programming languages, such as packages, types, functions and variables. An overview of the central entity types are shown in Fig. 2. Languages in Storm are encouraged to use and extend these entity types where possible so that different languages and extensions are able to communicate with each other without explicit knowledge of each other. Storm follows this convention by exposing all of its

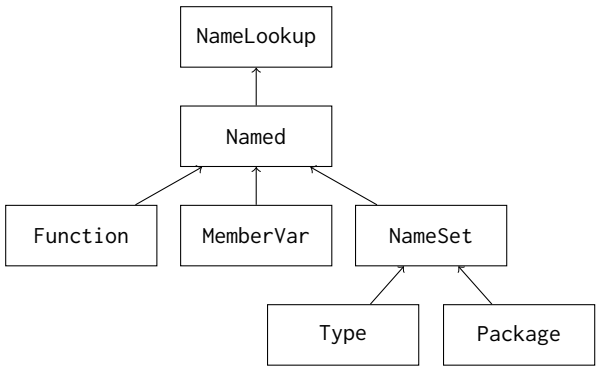

Figure 2: Overview of the relation between a few central entity types in Storm. The entity types `Named` and `NameSet` provide the structure of the name tree itself. The remaining entity types define common concepts that can be used by languages to interact.

functionality through standard entity types in the name tree. An entity type is simply a class that inherits from `Named`. The term *entity type* is used to highlight that it is the type of an entity in the name tree.

The three most important entity types provided by Storm are perhaps the `Type`, `Function`, and `MemberVar` entity types. Together, they form the core of the type system used by Storm to expose its functionality to languages in the system. Storm is statically typed using a type system that is similar to the type system in Java. Types may contain both variables and functions, that are considered members of the type. Single inheritance between types is supported, and virtual dispatch is used for member functions in order to achieve polymorphism. Similarly to Java, Storm differentiates between class-types and value-types. The former have by-reference semantics while the latter have by-value semantics. There are no primitive types in Storm. These are instead provided as value-types by the standard library. The type system in Storm also supports actor-types to manage concurrency, but this is outside the scope of this paper. The name tree itself is exposed to the system using this type system.

Named entities of the `Type` entity type describe types in the system. As can be seen in Fig. 2, the `Type` entity type inherits from the `NameSet` entity type. This makes it possible to add member functions and member variables to `Type` entities by adding them as children in the name tree. The `Type` entity keeps track of its contents to generate additional metadata. In particular, it uses the contained `MemberVar` entities, that each represent a member variable, to compute the in-memory layout of instances of the type. It also examines the contained `Function` entities in order to determine which member functions require vtable-based dispatch, and adds such logic if necessary. Of course, `Function` entities can be used outside of types to create non-member functions.

4.4 Populating the Name Tree

The `Package` entity type (see Fig. 2) is another important entity type provided by Storm. As it inherits from the `NameSet` entity type it is also used to structure the name tree hierarchically. In addition

to providing structure, Package entities are also responsible for populating the name tree. Each Package entity is associated with a directory in the file system. The package is then responsible for populating itself with named entities that correspond to the contents of the directory. The package will create a new Package entity for each subdirectory. It will also attempt to load the files in the directory by creating a *reader* for each file type. Readers are created by calling a function with the name `lang.<ext>.reader(...)` to create a reader for each file type that is found (`<ext>` is replaced with the file extension). The created reader(s) are then asked to create named entities that correspond to the contents of the file. These are then inserted into the package.

Since the Package searches the name tree to find a suitable reader, it is possible to add support for new file types by simply adding a named entity with the appropriate name to the name tree. Furthermore, the Package entity does not assume that the reader function is implemented in any particular language. Any language can be used as long as it produces entities that inherit from the standard Function entity type. Finally, it is worth mentioning that the system does not impose any restrictions as to how a reader interprets the files it is asked to load. In particular, files are not limited to text. For example, Storm provides readers that load dynamic libraries (i.e., .dll and .so-files) using this mechanism.

4.5 Generators

To allow implementing generic types (e.g., Array<T>), the name tree allows dynamically creating entities in response to name lookups in the name tree. This is done by adding a *generator* to a NameSet entity in the name tree. A generator is a function that is associated with a name. In contrast to named entities, the name of a generator only consists of a string. A generator does not have a list of parameters.

The generator is then called by the NameSet entity whenever the NameSet entity fails to find an entity that matches a requested name. For example, assume that a language requests a named entity with the name Array(int) from a NameSet. The NameSet first attempts to find a named entity with the specified name. If none exists, it calls any generators with the name Array to try to create a named entity that matches. The generators are given a list of the parameters that were requested, and from that information they may chose to generate a named entity. If this happens, the created named entity is added to the NameSet and returned to the language.

5 EXAMPLES OF EXTENSIBILITY

This subsection illustrates the extensibility made possible by the name tree and the standard entity types in Storm through three examples. As we will see, the languages and language extensions in the examples are able to utilize the name tree to preserve type-safety across language borders, and even introduce new concepts that are properly type-checked.

5.1 Grammars in the Syntax Language

As mentioned in Section 2.3, the Syntax Language stores non-terminals and productions in the name tree. This has two main benefits. First, it makes it possible to treat grammars as modules, and to selectively enable the grammar that implements a syntax extension by importing a package from the name tree. Secondly, it makes the grammar available to other languages in the system which, as we shall see, makes it possible to implement type-safe parse trees and syntax transforms.

```
1 Str Word();
2 Word => x : "[A-Za-z]+" x;
3
4 Array<Str> Words();
5 Words => Array<Str>()
6   : Word -> push - (" +" - Word -> push)*;
```

Listing 1: Simple grammar in the Syntax Language for parsing a series of words separated by whitespace.

To illustrate the benefits of this approach we will use the grammar in Listing 1 that matches words surrounded by whitespace. The syntax and semantics of the Syntax Language have been described in further detail in an earlier paper [16], so we will only cover what is used in the example in this paper.

The grammar defines two non-terminals Word and Words on lines 1 and 5 respectively. The definitions have the same form as function declarations in C since they define the parameter- and return types of the associated transform functions. In this case, it is possible to transform a match of the Word non-terminal into a string, and a match of the Words non-terminal into an array of strings.

The grammar also defines two productions. The production on line 2 defines a production for the non-terminal Word that simply matches one or more letters. The definition also describes that when the associated node in the parse tree is transformed, the matched text should be bound to x, and that x should be returned. The second production on lines 5 and 6 is a bit more complex. As can be seen on line 6, it first matches the non-terminal Word, followed by the contents of the parentheses repeated zero or more times. Each repetition of the parentheses matches one or more spaces followed by the non-terminal Word. The production also defines how a match should be transformed into an array of strings. In this case, the expression after the big arrow on line 5 (=>) states that an array should be created. The small arrows on line 6 (->) then state that the corresponding matches of the Word non-terminal shall be added to the array by calling the push member function.

```
1 Array<Str> parse(Str input) {
2   Parser<Words> p;
3   p.parse(input, Url());
4   Words tree = p.tree();
5   Array<Str> result = tree.transform();
6   return result;
7 }
```

Listing 2: Code in Basic Storm that uses the grammar in Listing 1 to parse a string.

Both non-terminals and productions are stored in the name tree as named entities that inherit from the Type named entity. It is therefore possible to use these entities from other languages that are unaware of the representation used by the Syntax Language.

To illustrate this, consider the function `parse` in Listing 2 that uses the syntax from Listing 1 to parse strings.

The function first creates an instance of the type `Parser(Words)` on line 2 (written as `Parser<Words>` in Basic Storm). The parameter to the type specifies the starting non-terminal of the grammar. As we shall see, this allows the specialized `Parser` type to provide member functions that accurately reflect the type of the parse tree produced by the parser. The function then proceeds to parse the input by calling `parse` on line 3, before extracting the resulting parse tree on line 4. Note that the `tree` function returns an instance of the `Words` type, which is the name of the entity produced by the Syntax Language. Again, since the entity produced by the Syntax Language inherits from the `Type` entity type, Basic Storm is able to use it without special knowledge of the representation of grammars. Also note that the parameterization of the parser type makes the code type-safe. Since the parser knows it was instructed to start at the `Words` non-terminal, the `tree` function will always return an instance of the `Words` type as long as the parse was successful.

Finally, the function calls the `transform` member function of the created parse tree. This invokes the transform function of the node for the root of the parse tree that was defined in the Syntax Language. Again, since the Syntax Language represents the `transform` member as a standard `Function` entity, it is possible to call the function from Basic Storm. Furthermore, the type information provided by the entity makes it possible for Basic Storm to type-check the call as well. As such, we can see that this approach allows type-checking all steps of parsing, from input, through the parse tree, to the output of the transform functions. This approach also makes it possible to inspect the parse tree in a type-safe fashion, though it is not shown here for brevity.

Even though Basic Storm is unaware of the grammar representation used by the Syntax Language, it is possible to create language extensions that add this ability to Basic Storm. For example, Storm contains a library that allows using the Syntax Language to create recursive descent parsers. This library uses the fact that it is possible to retrieve the grammar from the entities produced by the Syntax Language, and uses this information to generate a faster but less powerful parser.

5.2 Type-safe SQL Queries

The SQL library in Storm is another example of a library that extends the standard entity types in the name tree. In contrast to the Syntax Language, the SQL library is implemented as a language extension to Basic Storm that allows embedding SQL statements as a DSL for database access. The library also allows declaring the structure of the tables in the database, which allows the library to type-check SQL queries ahead of time and makes it possible to access the result from queries in a type-safe way. Apart from bindings to database drivers, the library is implemented entirely in Basic Storm and the Syntax Language.

Listing 3 contains an example of the SQL library in Basic Storm. Line 1 contains a `use` statement that imports the SQL library. In Basic Storm, `use` statements need to appear first in the file as they are parsed separately from the remainder of the file. Each `use` statement have two effects: first, it makes all names in the specified

```
1  use sql;
2
3  DATABASE BookDB {
4    TABLE authors(
5      id INTEGER PRIMARY KEY,
6      name TEXT
7    );
8    TABLE books(
9      id INTEGER PRIMARY KEY,
10     title TEXT,
11     author INTEGER
12   );
13 }
14
15 Array<Str> booksByAuthor(BookDB db, Str name) {
16   var result = WITH db: SELECT * FROM books
17     JOIN authors ON books.author == authors.id
18     WHERE authors.name == name;
19
20   Array<Str> titles;
21   for (row in result) {
22     titles.push(row.books.title);
23   }
24   return titles;
25 }
```

Listing 3: Using the SQL library in Basic Storm to extract information from a small SQL database.

package visible without specifying the full name (like `import` statements in Java). Secondly, it makes the productions in the package visible when the remainder of the file is parsed. In this case, the productions in the `sql` library will be visible, which makes it possible to declare databases and execute queries. Note that this mechanism makes it possible to add productions to any non-terminal in the host language. In this case, the SQL extension adds a top-level declaration (the `DATABASE` keyword on line 3), and an expression (`WITH` on line 16).

Lines 3–13 contain a database declaration that defines the structure of the database that the program expects to use. The database contains two tables, `authors` (lines 4–7) that stores authors to the books in the database, and `books` (lines 8–12) that stores books, each with a single link to the author table.[4]

Similarly to the Syntax Language, the SQL library stores a database declaration as a named entity that inherits from the `Type` entity. In Listing 3, the entity is named `BookDB`. This type represents a *typed connection* to a database. When a typed connection to a database is established, the library verifies that the contents of the database matches the declaration of the typed connection. Furthermore, whenever a typed connection is used, the library uses the database declaration available in the name tree to type-check the queries.

In Listing 3, the typed connection is used in the query on lines 16–18, which performs a join over the two tables. The start of the query (i.e., `WITH db:`) specifies that the database connection in the variable `db` should be used to perform the query. The library is

[4]We are aware that many books have multiple authors, but we wished to keep the example simple.

then able to type-check the query based on this information. In particular, it verifies that all tables and columns exist, and verifies that the types for the comparisons on lines 17 and 18 are the same. This is also done for the right hand side of the comparison on line 18, even though it refers to the variable `name` that is declared as a parameter in Basic Storm.

The library also generates an appropriate type for the result of the query to make it convenient to extract the result. In this case, the query produces an iterator that generates individual rows with the appropriate members. This can be seen on line 22, where the expression `row.books.title` is used to access the `title` column of the `books` table. Since `row` is a type with the appropriate members, Basic Storm is able to type-check both the existence of the column, and that the type is appropriate for the the call to `titles.push`.

5.3 Algebraic Effects

Another example of a library that extends the capabilities of Basic Storm is the experimental library that provides continuations through algebraic effects [12]. As we shall see, the library adds the ability to define type-safe effects, type-safe handlers for the effects, and syntax to guard code inside a handler. Apart from the ability to save and restore the execution stack of green threads, the library is implemented entirely in Basic Storm and the Syntax Language.

```
1  use experimental:effects;
2
3  effect fork()->Int;
4
5  handler ForkHandler(Int->Int) {
6    fork(), cont {
7      Int first = cont.call(1);
8      Int second = cont.call(2);
9      return first + second;
10   }
11 }
12
13 void main() {
14   Int result = with ForkHandler handle {
15     print("Before fork");
16     Int x = fork();
17     print("Fork returned: ${x}");
18     return x * 2;
19   };
20   print("Result: ${result}");
21 }
```

Listing 4: Implementation of a simple effect that behaves similarly to the `fork` system call.

Listing 4 contains an example that illustrates how the effect library is used. Line 1 imports the effects library to make the new syntax available, like in the previous example. The example then declares the `fork` effect on line 3. In this case, the effect accepts no parameters and returns an integer. The effect is stores as a named entity that inherits from the `Function` entity type in the name tree. This makes it possible to call the effect as if it was a regular function from Basic Storm.

Lines 5–11 defines a handler named `ForkHandler`. The first line specifies that the code in the handler block is expected to evaluate to an integer, and that the handler block itself will evaluate to an integer (`Int->Int`). On lines 6–10, the handler then defines how it handles the `fork` effect defined on line 3. The library is able to verify that the name actually refers to a handler, and that the parameter list matches by inspecting the entity in the name tree. The additional parameter, `cont`, is the name that will be bound to the continuation captured when the effect was called. The continuation is resumed twice in the handler, on lines 7 and 8. Again, the library uses the available type information to ensure that the continuation has the correct type.

```
1 Before fork
2 Fork returned: 1
3 Fork returned: 2
4 Result: 6
```

Listing 5: Output produced by the program in Listing 4.

The effect and the handler are then used in the `main` function on lines 14–19, which contains a `handle` block that wraps the code that should be affected by the effect. This code simply prints a message, calls the effect, prints another message and returns from the handler. However, since the handler resumes the continuation twice (on lines 7 and 8), the second print statement is executed twice. Furthermore, since the handler adds the results from the two executions, the handler block will evaluate to the sum of the two `return` statements on line 18. Thus, the program produces the output shown in Listing 4.

It is worth noting that the use of the keyword `with` in the effects extension (Fig. 4) and the SQL extension (Fig. 3) is coincidental. Neither Storm nor Basic Storm requires using particular keywords to trigger extensions. As mentioned previously, a syntax extension is able to add productions to arbitrary non-terminals in the host language or any other extension.

This example once again illustrates the flexibility of the name tree. In this case, the library adds two new entity types to the system. One corresponding to effects that inherits from the `Function` type entity, and another that corresponds to handlers that inherits from the `Type` entity type to make it possible for handlers to contain state. The latter also adds a set of handlers as a separate concept to the entity. The example also illustrates the the flexibility allowed by the runtime system in Storm.

6 EXAMPLES OF LEARNABILITY

As noted by Ghosh [7], one drawback with extensible languages and DSLs is that they make it necessary for developers to learn new DSLs and language extensions. As this involves learning new syntax and semantics, it usually requires more effort compared to learning a library. As we saw in the examples in Section 5, even the embedded DSLs provided by the SQL and effect libraries make the host language, Basic Storm, look like a different language to some extent.

Fortunately, the name tree is useful in this regard as well. Since the name tree is accessible to programs in Storm, it is possible to create tools that make it easier to learn the different libraries and languages in the system, and to debug issues in the system. The remainder of this section will describe two existing such tools. The

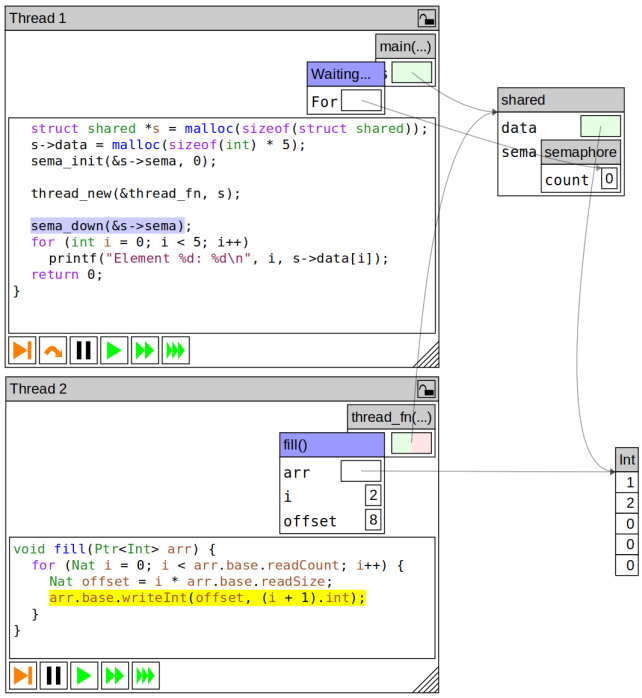

Figure 3: Screenshot of Progvis visualizing a program that consists of two threads. Thread 1 executes code written in C, while thread 2 executes code written in Basic Storm.

section will conclude with proposing an idea how the existing language server can be extended with capabilities that allow exploring and debugging language semantics based on existing source code.

6.1 Progvis: A Visual Debugger

Progvis is a visualization tool that was originally designed to visualize concurrent programs written in C [17]. However, since Progvis is implemented in Storm and operates on the representation in the name tree, it is not limited to programs written in C, but can visualize programs in any language in Storm, given that sufficient metadata is emitted by the language.

Figure 3 contains a screenshot of Progvis visualizing a program that consists of two threads. Thread 1 executes code written in C. This thread has created a data structure that it shared with thread 2. Thread 2 started executing code in C, but then called a function written in Basic Storm. As such, the box for thread 2 currently shows the Basic Storm code that is about to be executed. Note that the two threads share data even though they run code written in two separate languages.

As mentioned previously, Progvis is able to support multiple languages in this fashion by utilizing information available in the name tree. In particular, Progvis relies on the type-information in the name tree to traverse data structures, as well as the ability to extract and modify the generated code in the intermediate representation. When the user selects a file to visualize in the user interface, Progvis creates a package in the name tree and loads the program into the package using the standard reader facilities. Progvis then inspects the generated named entities in the name tree to find all `Function` entities. For each function, it extracts the generated code adds instrumentation to it, and replaces the code in the function entity with the modified version. After the instrumentation process, Progvis simply calls the `main` function in the program and relies on the instrumentation code to extract data from the running program. Any data structures used by the program can easily be traversed by relying on the data available in the corresponding `Type` entity that was generated by the compilation process.

Progvis does, however, need some effort from the visualized language in order to be able to provide a good visualization. First and foremost, the language needs to annotate stack-allocated variables in the intermediate representation with a name and a type. Only variables with annotations are shown to the user, since other variables are assumed to be unnamed temporary variables needed by the language. Secondly, the language needs to emit pseudo-instructions that map the intermediate code to locations in the source code. Progvis uses this information to determine when the program should be paused, and what area of the source code to highlight at that time. Lastly, a language may optionally provide custom visualizations of certain data structures to make them easier to see. For example, the C front-end provides a custom visualization of the semaphore in the data structure `shared` in order to hide implementation details from the user.

Finally, it is worth noting that Progvis is designed for novices. Its detailed visualization is therefore not suitable for debugging large systems with complicated data structures. Not because of inherent limitations in the approach, but because of limitations in the user interface. It does, however, illustrate the possibilities offered by the approach.

6.2 On-line Documentation

Each named entity in the name tree in Storm is able to store documentation about itself. This is used to implement on-line documentation in Storm. Much like in languages like Python, it is possible to launch Storm interactively and type `help <name>` to show the documentation for a particular named entity. An Emacs plugin is also available to allow Emacs to communicate with Storm using a custom protocol. This provides a richer interface where the user can browse the name tree by clicking hyperlinks rather than typing the names of entities. Both of these interfaces have proven useful not only for accessing documentation, but also for debugging language implementations, since both interfaces makes it possible to view the named entities generated by languages and language extensions.

6.3 Semantics of Language Constructs

While the system for browsing the documentation is useful to explore libraries through the name tree, it is not very useful to explore the meaning of constructs inside functions. For example, it is currently difficult to find the semantics of specific constructs in Basic Storm or extensions to Basic Storm.

To help users explore and learn the syntax of languages in the system, it would be good if it was possible to see how a piece of source code was parsed and what named entities implement the semantics. All of this information is available in the name tree in Storm. As previously mentioned (see Section 3), the system already

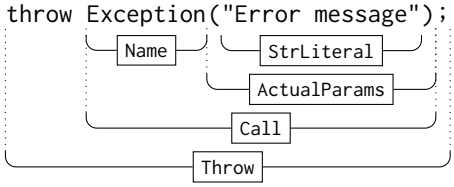

Figure 4: Illustration of how the meaning of a line of source code could be visualized using the information available in the name tree. Each box would be clickable and would refer to the named entity used in the corresponding transform function.

contains a language server that is able to interactively parse source code while it is being edited in order to provide syntax highlighting to the aforementioned Emacs plugin. As a side-effect, a parse tree is already available and could be visualized to the user upon request. Furthermore, most productions contain references to named entities as a part of the syntax transforms. The referred named entities are usually types or functions that implement the semantics of the matched language construct. As such, it is possible to link individual sub-strings in the source code to the named entities in the name tree that implement the semantics.

Figure 4 contains an illustration of a possible way to visualize this information. The original source code is displayed in the top of the figure. If the user requests the editor to show the semantics of the line, the editor would first move all lines below the current line downwards to create space for the visualization. The editor would then use the empty space to show the relevant part of the parse tree below the original source code, as is shown in Fig. 4. The visualization would be interactive and allow the user to click each box to show the documentation for the named entity referred to by the transform function associated to the production.

Even though the metadata is available, this functionality is currently not implemented in Storm. As such, there are a number of open questions regarding the practicality of the approach. Most importantly, the system likely needs to select a subset of the nodes in the parse tree to show. Otherwise the representation is likely to become too complex to be understandable. A good starting point is likely to only show productions that call a function or create an instance of a type in their syntax transform. This would produce the level of detail shown in Fig. 4, which is at least suitable for simple expressions.

7 RELATED WORK

There are a number of tools that allow introducing new syntax to a language. An early example is macros in Lisp. These were later refined in Racket by providing facilities like pattern-based macros [5]. Similarly, tools like ableJ [19] and ableC [10] provide extensibility to Java and C respectively. This is achieved by allowing the creation of modules of syntax that extend the syntax of the host language. The modules also contain semantics that transform the syntax into equivalent statements in the host language. SugarJ [3] is another tool that provides extensible syntax to Java similarly to ableJ. A notable difference between the two is that SugarJ is self-applicable, which makes it possible to use extensions in SugarJ when developing future extensions, much like in Storm.

There are also a number of tools, such as Xtext [4] and Spoofax [11], that focus on stand-alone DSLs rather than embedded ones. While these tools do allow embedding languages inside each other, this can only be done by the language designer when the DSL is created. As such, a user of the language is not able to select which extensions to use on a file-by-file basis, as is possible in Storm and the tools mentioned in the previous paragraph. Another similar system is Truffle on GraalVM [8]. GraalVM is itself a polyglot runtime that allows multiple languages to interact, and Truffle is a framework for creation of languages that allows composing languages. Riese et al. [14] have, however, stated that inter-language communication with complex datatypes is sometimes cumbersome and propose interface mappings to alleviate the problem.

One drawback with all of these tools compared to Storm is that they all eventually produce code in a host language (e.g., Java or C), or in a fixed shared representation. Since this representation is not extensible, like the one provided by name trees in Storm, it is difficult to introduce new concepts in a way that is easily consumable and further extended by other languages in the system. Furthermore, since language implementations are available through the name tree, new languages are able utilize existing languages as an intermediate step during the compilation process. For example, the Syntax Language does not emit code in the IR provided by the code generation library directly. Rather, it produces a Basic Storm AST and relies on Basic Storm to produce the final IR. This ability makes it possible to implement languages as a series of smaller passes, similarly to the ideas behind *nanopasses* [15]. The passes currently used by Storm are, however, larger than the proposed nanopasses.

Management of scope is also important when working with extensible languages. For example, Racket utilizes sets of scopes [6] to manage scope in macro expansions, and to simplify the implementation of hygienic macros for DSLs. While sets of scopes is not natively supported by the notion of scope described in Section 4.2, the mechanism is general enough to implement it if desired. Even without sets of scopes, the name tree allows explicit control of the scope in macro expansions in Basic Storm, for example.

Spoofax also provides a generic representation of scopes, called *scope graphs* [20]. As the name implies, a scope graph is a graph that describes the visibility of other entities in the system. The main difference to name trees is that scope graphs model visibility directly in the graph, while name trees model visibility through a traversal strategy. An advantage of the approach used by name trees is that a language is able to customize their view of the shared namespace to suit its needs, possibly modifying scoping rules from other languages if necessary.

When working with extensible languages and DSLs it is also important to consider the developer experience. For an acceptable developer experience, a language should provide at least basic syntax highlighting, automatic indentation, and a convenient way to build and run the program. The aforementioned tools Xtext [4] and Spoofax [11] are both *language workbenches*, and are therefore also able to generate tooling in the form of plug-ins for common IDEs. TigersEye [2] is a tool for embedded DSLs, similar to SugarJ, ableJ, and ableC, that is also able to provide tooling in the form of an IDE

plugin that extends the IDE's existing support for the host language. Similarly, Storm is able to provide syntax highlighting and automatic indentation through an integrated language server that relies on the named entities available in the name tree. Furthermore, even though Storm provides a large degree of flexibility, the entire build process is managed by Storm itself. As such, running programs in Storm is often as easy as running `storm <path>` where `<path>` is the name of a directory that contains the application.

8 CONCLUSION

In this paper, we have introduced the *name tree* that Storm uses to provide an extensible environment for implementing extensible languages. The name tree itself provides a uniform mechanism for storing and retrieving *named entities* that may represent any element that has a name in a language. To facilitate language interoperability, the system provides a number of *standard entity types* that define a common interface for concepts like functions and types. As we have seen, languages or language extensions may extend the standard entity types in order to represent new concepts in the name tree. When the standard entity types are extended, existing languages are able to use at least some part of the new concepts, even though the existing languages are not aware of the new representation. This also makes it possible to introduce new concepts to the system in a type-safe way. For example, as shown in Sections 5.2 and 5.3, it is possible to create language extensions that introduce type-safe SQL queries and algebraic effects as libraries.

Since Storm is language-agnostic and provides the ability to execute compiled code in-process, it is possible to use any language in the system to implement languages and language extensions, as long as the language produces entities that are compatible with the standard entity types. With some care to avoid cyclic dependencies, it is even possible to implement languages partially in themselves, or to create extensions that simplify the development of future extensions. These possibilities do, however, not complicate the build process. Since Storm utilizes lazy compilation, running a Storm program is often as easy as running `storm <path>`, where `<path>` is a directory that contains the application. The name tree is also able to help the developer experience. As shown in Section 6, the introspection allowed by name trees allows creating useful development tools for exploring syntax, visual debugging, and more.

REFERENCES

[1] Richard Brooksby and Nicholas Barnes. 2002. *The Memory Pool System.* Technical Report. https://www.ravenbrook.com/project/mps/doc/2002-01-30/ismm2002-paper/ismm2002-letter.pdf

[2] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. 2013. Incremental concrete syntax for embedded languages with support for separate compilation. *Science of Computer Programming* 78, 6 (2013), 615–632. https://doi.org/10.1016/j.scico.2012.12.002 Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[3] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

[4] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

[5] Matthew Flatt. 2011. Creating Languages in Racket. *Queue* 9, 11, Article 21 (Nov. 2011), 15 pages. https://doi.org/10.1145/2063166.2068896

[6] Matthew Flatt. 2016. Binding as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 705–717. https://doi.org/10.1145/2837614.2837620

[7] Debasish Ghosh. 2011. DSL for the Uninitiated. *Queue* 9, 6, Article 10 (June 2011), 12 pages. https://doi.org/10.1145/1989748.1989750

[8] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, PA, USA) *(DLS 2015)*. Association for Computing Machinery, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[9] Michael Hicks and Scott Nettles. 2005. Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (nov 2005), 1049–1096. https://doi.org/10.1145/1108970.1108971

[10] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

[11] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

[12] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[13] Joan Gerard Rekers. 1992. *Parser generation for interactive environments.* Ph. D. Dissertation. Universiteit van Amsterdam.

[14] Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2020. User-defined interface mappings for the GraalVM. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(Programming '20)*. Association for Computing Machinery, New York, NY, USA, 19–22. https://doi.org/10.1145/3397537.3399577

[15] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/1016850.1016878

[16] Filip Strömbäck. 2018. Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. Association for Computing Machinery, New York, NY, USA, 60–64. https://doi.org/10.1145/3276604.3276982

[17] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. Pilot Study of Progvis: A Visualization Tool for Object Graphs and Concurrency via Shared Memory. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/3511861.3511885

[18] Pablo Tesone, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. 2018. Dynamic Software Update from Development to Production. *Journal of Object Technology* 17, 1 (Nov. 2018), 1:1–36. https://doi.org/10.5381/jot.2018.17.1.a2

[19] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. 2007. Attribute Grammar-Based Language Extensions for Java. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 575–599.

[20] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.EVCS.2023.32